Beyond the Input Stream: Making Text Entry Evaluations More Flexible with Transcription Sequences

Mingrui "Ray" Zhang
The Information School
DUB Group
University of Washington
Seattle, WA, USA 98195
mingrui@uw.edu

Jacob O. Wobbrock
The Information School
DUB Group
University of Washington
Seattle, WA, USA 98195
wobbrock@uw.edu

ABSTRACT

Method-independent text entry evaluation tools are often used to conduct text entry experiments and compute performance metrics, like words per minute and error rates. The input stream paradigm of Soukoreff & MacKenzie (2001, 2003) still remains prevalent, which presents a string for transcription and uses a strictly serial character representation for encoding the text entry process. Although an advance over prior paradigms, the input stream paradigm is unable to support many modern text entry features. To address these limitations, we present *transcription sequences*: for each new input, a snapshot of the entire transcribed string unto that point is captured. By comparing adjacent strings within a transcription sequence, we can compute all prior metrics, reduce artificial constraints on text entry evaluations, and introduce new metrics. We conducted a study with 18 participants who typed 1620 phrases using a laptop keyboard, on-screen keyboard, and smartphone keyboard using features such as auto-correction, word prediction, and copy/paste. We also evaluated non-keyboard methods *Dasher*, *gesture typing*, and *T9*. Our results show that modern text entry methods and features can be accommodated, prior metrics can be correctly computed, and new metrics can reveal insights. We validated our algorithms using ground truth based on cursor positioning, confirming 100% accuracy. We also provide a new tool, *TextTest++*, to facilitate web-based evaluations.

CCS Concepts
• **Human-centered computing~Text input**
• Human-centered computing~Laboratory experiments
• Human-centered computing~Empirical studies in HCI

Keywords
Text entry evaluation; text entry metrics; words per minute; error rates; presented string; transcribed string; input stream; transcription sequence.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.
UIST '19, October 20–23, 2019, New Orleans, LA, USA
© 2019 Association for Computing Machinery.
ACM ISBN 978-1-4503-6816-2/19/10…$15.00
https://doi.org/10.1145/3332165.3347922

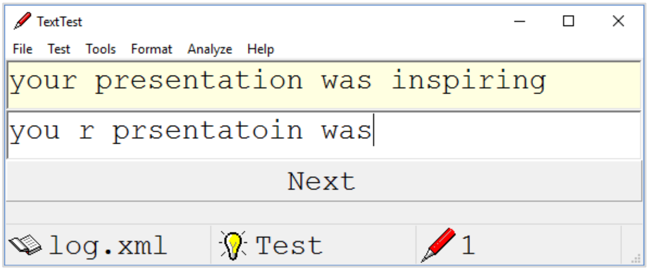
TextTest
File Test Tools Format Analyze Help
your presentation was inspiring
you r prsentatoin was
Next
log.xml
Test
1

Figure 1. A typical text entry transcription task, shown in the Windows-based *TextTest* evaluation tool [27]. A presented string is displayed above a textbox that receives the string entered by the participant.

INTRODUCTION

Text entry remains fundamental on most computing platforms, from desktops to tablets to game consoles to smartphones. Increasingly, the need for text entry extends to new platforms, such as interactive tables [4], smartwatches [28], and augmented reality [29]. As a result, researchers, developers, and product innovators still regularly create new text entry methods.

When seeking to quantify the performance of their new methods, creators can benefit from pre-existing testbeds, rather than having to build their own evaluation tools. Such pre-existing testbeds must be "method independent," working without any feature-specific knowledge of the text entry methods they evaluate. Therefore, such tools receive text and compute metrics (*e.g.*, words per minute [12], various error rates [22], and more) without knowing the mechanisms by which that text is produced. (We refer to this as the "black box" consideration, and discuss it below).

To compute method-independent metrics, evaluation testbeds must artificially constrain the text entry evaluation process. Measuring text entry error rates presents a particular challenge, and is a major reason for artificial constraints, because we must infer a user's intention in order to detect deviations from it [3,13,27]. The prevalent evaluation paradigm addressing this need is that of Soukoreff & MacKenzie [21,22], which, in each text entry trial, presents a string that a participant transcribes (Figure 1). The accompanying model encoding a participant's text entry process is called the *input stream (IS)*, which is a strictly sequential record of each character entry or BACKSPACE. Unfortunately, the *IS* model cannot accommodate many modern text entry behaviors, including using the mouse or arrow keys to position the cursor, text highlighting and replacement, auto-correction, word prediction, undo, and copy/paste, to name a few. Traditionally, the only means of error correction in the *IS* paradigm is BACKSPACE, and only then from the end of the currently entered text.

In this work, we present a new underlying model that supersedes the *IS* model for general-purpose method-independent character-level text entry evaluation. Specifically, we present an approach that replaces the input stream with *transcription sequences*, or "*T*-sequences" for short. In brief, a *T*-sequence is a sequence of snapshots of the entire transcribed string after each text-changing action is taken by the user. Every pair of successive snapshots are then compared to compute character-level text entry metrics. We show that, unlike the *IS* paradigm, the *T*-sequence paradigm can handle the modern text entry features that have been disallowed thus far. We validated our measurement algorithms using ground truth cursor-position information, confirming 100% accuracy. We also show that *T*-sequences not only accommodate prior metrics from the *IS* paradigm, but also enable new metrics. Finally, we offer a web-based successor to the *TextTest* desktop application [27] (Figure 1) called *TextTest*++,[1] which encapsulates our approach and enables text entry innovators to study their inventions on any platform or device capable of running a web browser.

The contributions of this work are: (1) The detailed elucidation of a new general-purpose method-independent model of the text entry process based on *T*-sequences, superseding the *IS* model; (2) New algorithms for computing text entry metrics, both extant and novel; (3) Empirical results from a study of 18 participants confirming the ability of our model to handle modern text entry behaviors; (4) An evaluation of *T*-sequences with non-keyboard techniques *Dasher* [25], *gesture typing* [8,30], and *T9* [11]; and (5) The web-based *TextTest*++ evaluation testbed capable of conducting and analyzing text entry studies. This work will be useful to text entry method creators wishing to evaluate their new methods without imposing undue constraints.

BACKGROUND AND RELATED WORK

To appreciate the current work, it is important to understand the history of text entry evaluation and the challenges faced. We divide this section into three parts. The first offers a general background on method-independent text entry evaluations, situating this work among its predecessors. The second offers a more detailed look at calculating error metrics in the input stream paradigm of Soukoreff & MacKenzie [21,22]. The third describes other error-related text entry metrics that have arisen in the literature.

Method-Independent Text Entry Evaluation

In the 1990s, method-independent text entry transcription experiments were highly constrained for the purpose of error measurement. Some studies simply ignored errors [10]. Other studies prohibited erroneous characters from appearing [23]. Yet other studies disabled all error correction [15]. These latter two approaches meant that single-character mistakes could result in long error "chunks" [16], forcing participants to resynchronize with the presented string.

In 2001, seminal work by Soukoreff & MacKenzie [21] began to loosen these constraints by using the Levenshtein minimum string distance algorithm [9] to calculate errors based on the edit distance between two strings.[2] BACKSPACE was now allowable as the sole means of error correction, and transcribed characters no longer had to "line up" with the presented characters directly above them in order to avoid being counted as errors. Participants could enter text freely without having to resynchronize after an inserted or omitted character. Soukoreff & MacKenzie's influential 2003 paper [22] showed how to calculate error rates in this paradigm. (We elaborate on these calculations in the next subsection.)

The underlying model in Soukoreff & MacKenzie's work [21,22], and much that followed (e.g., [13,27]), was that of a text entry *input stream (IS)*, where character entries and BACKSPACEs (<) were recorded sequentially, like so:

thr<**e qu**ck<<**ick br**wn<<**o**n<**wn**

The resulting transcribed string from the *IS* above is "the quick brown," with six BACKSPACEs encoded as error corrections during entry. The *IS* not only contains all information necessary to extract the final transcribed string, it also contains all dynamic information about the text entry process that created it. From this information, Soukoreff & MacKenzie [22] defined three separate error rates: (i) uncorrected errors, for those remaining in the final transcribed string; (ii) corrected errors, for any characters backspaced during entry; and (iii) total errors, for their sum. Because the *IS* model is strictly serial, only able to append edits to its right-hand side, numerous editing restrictions are imposed in this paradigm: (1) BACKSPACE is the only error correction mechanism allowed; (2) the text cursor must always remain at the end of the string entered thus far—no mouse or arrow keys can be used to move the text cursor; (3) selecting-and-replacing text is not allowed; and (4) auto-correction is not feasible. Although researchers can develop custom analyses and evaluation tools for individual text entry methods based on method-specific knowledge, method-*independent* evaluations based on the *IS* model cannot accommodate many modern text entry behaviors. Our work remedies these limitations, and offers a method-independent evaluation paradigm and platform-independent evaluation testbed.

Error Rates in the Input Stream Paradigm

As mentioned above, Soukoreff & MacKenzie [21,22] enabled much less constrained text entry evaluations than what had come before, while still being able to calculate

[1] Available at http://depts.washington.edu/acelab/proj/texttestpp/

[2] The edit distance is the minimum number of character insertions, deletions, or substitutions required to turn one string into another.

error rates. Their paradigm relied on the building of an *IS* containing entered characters and BACKSPACES. As an example, consider the following presented phrase (*P*) and final transcribed phrase (*T*):

P: `the quick brown fox`
T: `the quick brqn fox`

Soukoreff & MacKenzie utilized the minimum string distance (MSD) [21] to align *P* and *T* and compute the minimum number of insertions, deletions, or substitutions necessary to equate them. However, by only considering *P* and *T*, all dynamic in-process information was lost. Therefore, Soukoreff & MacKenzie used the *IS* to classify each character into one of four classes [22]: Correct (*C*), Incorrect and Not Fixed (*INF*), Incorrect and Fixed (*IF*), and Fixes (*F*). For example:

IS: $\overbrace{\texttt{the qu}}^{C}\overbrace{\texttt{oc}}^{IF}\underbrace{\texttt{<<}}_{F}\overbrace{\texttt{ick br}}^{C}\underbrace{\texttt{q}}_{INF}\overbrace{\texttt{n fox}}^{C}$

All characters in *T* belong to the *C* or *INF* classes. As stated, *C* contains the correct characters in *T*, and *INF* is computed using the aforementioned MSD statistic:

$$C = \mathrm{MAX}(|P|, |T|) - \mathrm{MSD}(P, T) \quad (1)$$

$$INF = \mathrm{MSD}(P, T) \quad (2)$$

The *F* class contains editing keystrokes, which, in Soukoreff & MacKenzie's *IS* paradigm, is only BACKSPACE. Finally, *IF* contains any characters that are erased, whether they were initially correct or not. Both *F* and *IF* are counted by making a backwards scan over the *IS*.

With these classifications in place, three crucial error rate metrics can be calculated [22]: the uncorrected error rate (*UER*), the corrected error rate (*CER*), and their sum, the total error rate (*TER*), as follows:

$$UER = INF / (C + INF + IF) \quad (3)$$

$$CER = IF / (C + INF + IF) \quad (4)$$

$$TER = (INF + IF) / (C + INF + IF) \quad (5)$$

As noted, although this work was a great advance over prior paradigms, the *IS* paradigm still had a major limitation, namely that all editing was sequential and only could occur at the end of the currently entered text. This limitation rules out numerous features, like using the mouse or arrow keys to position the cursor, copy/paste, undo, and auto-correction.

Other Error-Related Metrics in Text Entry

Researchers have developed additional error-related metrics in text entry research, most based within the *IS* paradigm. One project looked at character-level error rates in particular. MacKenzie & Soukoreff [13] formed optimal alignments between *P* and *T* that represented the various ways the minimum string distance could be achieved. For example, there are two optimal alignments for *P* = "optimal" and *T* = "optiacl", each reflecting MSD = 2:

P_1: `optimal`
T_1: `optiacl`

P_2: `optima-l`
T_2: `opti-acl`

In the alignment pairs above, substitution errors occur at character mismatches; omission errors occur where *T* has a '-'; and insertion errors occur where *P* has a '-'. By weighting these errors by the number of alignments in which they occur, individual substitution, omission, and insertion error rates can be calculated for each character.

Character-level analyses were extended to the *IS* model by Wobbrock & Myers [27]. They created an extensive set of character-level error rates and ratios concerning substitutions, omissions, and insertions.

Other error-related metrics such as cost per correction [2,5] and word error rate [6] have been introduced. However, these metrics also rely on the sequential *IS* paradigm, and therefore are similarly constrained.

Word-level metrics have been utilized in some recent text entry studies [29,31]. However, these studies only compared the final transcribed string (*T*) with the presented string (*P*), not examining the dynamic text entry process (*i.e.*, no corrected error rates). Similarly, some studies [24,28] used a character error rate, which just uses MSD(*P*, *T*), again revealing nothing about the dynamic text entry process.

We are not the first to log snapshots of successive transcribed strings rather than just keypresses, or to provide a web-based evaluation tool. The capable *WebTEM* tool by Arif & Mazalek [1] seems to employ a similar logging paradigm to ours; however, their paper does not explore any properties of transcription sequences or describe any algorithms that operate on them, reporting only that, "WebTEM detects all input based on the events in the input area" (p. 417). No information is given about how WebTEM computes error rates or other metrics.

Ruan *et al.* [19] also logged successive transcriptions; however, they utilized method-*specific* knowledge for analysis and did not formalize general algorithms for operating on transcription sequences, as we do here.

A METHOD-INDEPENDENT ABSTRACTION OF TEXT ENTRY EVALUATION

As discussed above, any pre-existing text entry evaluation testbed must, by definition, operate without any method-specific knowledge of the text entry method it evaluates. As the specific features of the text entry method are unknown, the method can be considered a black box (Figure 2). An abstraction containing three parts is therefore implied: (1) the user's actions on the text entry method, (2) the black box text entry method itself, and (3) the resulting text output that enters the method-independent evaluation testbed. In short, (1) acts on (2) to produce (3).

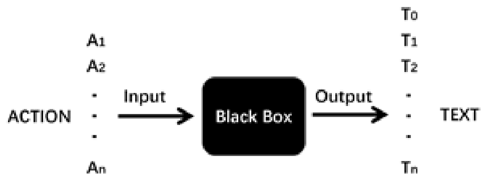

Figure 2. An abstraction of the method-independent text entry evaluation process. A user takes actions on a black box text entry method, which generates text as output, which is received by an evaluation testbed for logging and analysis.

After each input action that alters the current transcription, rather than appending a character (or BACKSPACE) to the end of an input stream, the *entire* transcription at that moment is captured. This transcription, denoted T_i, is one entry in a sequence of transcriptions, or "*T*-sequence," captured after each action. Below is an example for entering "computer".

Transcription (T_i)		Action (A_i)	
T_0:	<null>	A_0:	<begin>
T_1:	`c`	A_1:	*insert* `c`
T_2:	`co`	A_2:	*insert* `o`
. . .		. . .	
T_8:	`computer`	A_8:	*insert* `r`
T_9:	`computerr`	A_9:	*insert* `r`
T_{10}:	`computer`	A_{10}:	*delete* `r`

In the example above, the last "r" is typed twice by mistake and corrected. Each transcription T_i for $i > 0$ beyond the null transcription (T_0) is the result of a corresponding action A_i. An action should be thought of more broadly than just a concrete physical movement like "typing a key" or "making a stroke gesture;" rather, it is *any* action that transforms T_{i-1} into T_i. The next subsection provides an abstraction of actions. We build on this abstraction when analyzing *T*-sequences to extract text entry metrics.

An Abstraction of Actions

In our era of ubiquitous computing, there are any number of actions that might change text: typing a key, touching a screen, making a gesture, rotating a watch bezel, or typing CTRL + BACKSPACE on Windows to delete an entire word. Our black box abstraction only considers before-and-after text transcriptions resulting from each action. It only knows what the method-independent changes to text are, not the method-specific mechanisms *by which* the user brought about those changes. (Of course, the same was true of the input stream paradigm, but it could only handle single character serial entry and removal actions.) Formally, we classify each action A_i that turns transcription T_{i-1} into T_i as one of three classic types [13,27], based *only* on the changes to successive strings, not from any method-specific insights.

Insertion. An insertion is an action that adds one or more characters anywhere within or to either end of the current text, without removing any of that text. An insertion action is parameterized with two values: a zero-based start index where the insertion occurs (index zero is before the first character), and the string to be inserted. For example:

T_{i-1}: `All oads`
T_i: `All roads`

In this example, action A_i would be an insertion annotated with (4, "r"), meaning it starts at zero-based index 4 and the inserted string is "r".

Deletion. A deletion is an action that removes a substring anywhere within or at either end of the current text, without modifying other parts of that text. A deletion action is parameterized with two values: a zero-based start index at which the deletion occurs, and the number of characters forward from that point that are deleted. For example:

T_{i-1}: `All roads`
T_i: `All ads`

Above, action A_i would be a deletion annotated with (4, 2), meaning it starts at zero-based index 4, and two characters are deleted ("ro").

Substitution. A substitution is an action that composes a deletion and an insertion into one action. In a substitution, a substring is removed simultaneously as a new string, not necessarily of the same length, is inserted. A substitution action is parameterized with three values: a zero-based start index at which the deletion occurs, the number of characters forward from that point that are deleted, and the new string to be inserted. For example:

T_{i-1}: `All road lead to Rome`
T_i: `All paths lead to Rome`

In this example, action A_i would be a substitution annotated with (4, 4, "paths"), meaning it starts at zero-based index 4, deletes four characters ("road"), and inserts "paths".

For insertions or deletions, changes must happen in only one contiguous place in the text. If changes happen in multiple non-contiguous places simultaneously, we consider the change a substitution. For example, if T_{i-1} = "all roads" becomes T_i = "ball broads" by inserting a "b" before both "all" and "roads", we consider the change a single substitution. Other special cases like transportations, where chunks of text are moved to another place (*e.g.*, via drag-and-drop, a method-specific action), are also substitutions.

With this level of abstraction established, we are now ready to describe the transcription sequence paradigm and our algorithms for extracting text entry metrics from it.

THE TRANSCRIPTION SEQUENCE PARADIGM

We formalize the transcription sequence, or "*T*-sequence," paradigm using examples, and show how we infer actions from *T*-sequence changes. We also show how we can extract the old input stream (*IS*) from *T*-sequences that permit it.

Character Classes

Within the *T*-sequence paradigm, we reuse three of the four Soukoreff & MacKenzie character classes [22]: Correct (*C*), Incorrect and Not Fixed (*INF*), and Incorrect and Fixed (*IF*). We discard Fixes (*F*) because our black box abstraction and action definitions neither need nor permit method-specific information on the nature of fixes. To recap:

C – Correct characters in the final transcribed text, *T*.

INF – Minimum string distance (*MSD*) between *P* and *T*.

IF – All deleted characters in the entire sequence, T_0 to *T*.

By using these three classes, we can continue to calculate the uncorrected (*UER*), corrected (*CER*), and total (*TER*) error rates, Eqs. (3)-(5), above.

An Example of a Complete Transcription Sequence

Consider the following *T*-sequence representing the entry of *P* = "All roads lead to Rome". Note that whatever hypothetical text entry method is being used here, it has capabilities that enable it to go beyond just single-letter entry. In the Action column, "*I*" is insert, "*D*" is delete, and "*S*" is substitute.

Transcription (T_i)	Action (A_i)
T_0: <null>	A_0: <begin>
T_1: `A`	A_1: *I*(0, "A")
T_2: `All`	A_2: *I*(1, "ll")
T_3: `All Rome`	A_3: *I*(3, "_Rome")
T_4: `All roads`	A_4: *S*(4, 4, "roads")
T_5: `All roads l`	A_5: *I*(9, "_l")
T_6: `All roads let`	A_6: *I*(11, "et")
T_7: `All roads le`	A_7: *D*(12, 1)
T_8: `All roads lead to`	A_8: *I*(12, "ad to")
T_9: `All roads lead to rome`	A_9: *I*(17, "_rome")
T_{10}: `All roads lead to Rome`	A_{10}: *S*(18, 4, "Rome")

Note that all of "rome" was replaced by "Rome" with the final action (A_{10}), perhaps by a whole-word paste operation, auto-correction, or a spell-checker menu selection. The above example yields character classes as follows:

- *C* = {"All roads lead to Rome"}
- *INF* = { }
- *IF* = {"Rome", "t", "rome"}

Using Eqs. (3)-(5), the uncorrected error rate (*UER*) is 0.00%, the corrected error rate (*CER*) is 29.0%, and the total error rate (*TER*) is therefore 29.0%.

Inferring Actions from a Transcription Sequence

Consistent with our black box abstraction and our goal to improve method-independent approaches to text entry evaluation, we do not have ground truth information as to the inputs performed by the user or the actions that transform one transcription (T_{i-1}) into the next (T_i). We simply see the sequence of transcribed strings, and infer actions from successive transcriptions. For example:

T_{i-1}:	`Thai`	
T_i:	`Thanks`	A_i: ???

In this example, the user might have used auto-correction, in which case A_i is *S*(0, 4, "Thanks"). Or, the user might have selected the "i" and pasted "nks", which would be *S*(3, 1, "nks"). To know the truth of A_i, one would have to build a textbox capable of receiving *method-specific* signals and know how to interpret them.

Instead, it is possible to infer actions from changes between consecutive T_{i-1} and T_i. The rationale is that in most character-level input methods, characters only change over a contiguous range, not at multiple simultaneous disjoint indices; thus, it is sufficient to get the character change information. Doing so works on any platform, as all textbox widgets provide a property to inspect their text. We can then build a testbed for text entry evaluation that remains independent of any specific text entry method's features.

To infer the most likely actions taken given T_{i-1} and T_i, we created an algorithm called INFER-ACTION. The algorithm finds the minimum modification necessary to turn transcription T_{i-1} into T_i. In the example above, the algorithm would favor *S*(3, 1, "nks") over *S*(0, 4, "Thanks") because only one character, the "i", is changed in the former.

Our INFER-ACTION algorithm generally works as follows: Given the two strings T_{i-1} and T_i, it first compares them from their beginnings, stopping when it finds a mismatch at index p_1. Then it compares the strings from the end, again stopping when it finds a mismatch, now at p_2. Based on the relationship of p_1 and p_2, the algorithm determines whether the change is an insertion, deletion, or substitution. For example, if p_1 equals the length of T_{i-1}, p_2 equals the length of T_i, and T_i is longer than T_{i-1}, the action A_i inserted ($p_2 - p_1$) characters at the end of T_{i-1}.

Note that our INFER-ACTION algorithm is triggered only when there are changes in the text, *i.e.*, when T_{i-1} becomes T_i. If a user presses the CAPS LOCK key, drag-selects text with the mouse, or moves the text cursor with the arrow keys (all of which are method-specific actions), our algorithm would not consider a change, as the entered text remains unchanged. Evaluators wishing to go beyond quantifying general text entry performance to understanding *method-specific* behaviors (*e.g.*, the number of times CAPS LOCK was pressed on a keyboard) would need to build custom testbeds to capture such metrics, as they have always had to do.

To verify the correctness of our INFER-ACTION algorithm, we obtained ground truth in our study (explained below) by monitoring the text cursor movements with the JavaScript textbox properties `selectionStart` and `selectionEnd`. This information is sufficient for ground truth because when text changes, the text cursor appears at the end of the latest change. Our results show that INFER-ACTION, working only with *T*-sequence string pairs as described above, correctly inferred 100% of all actions in our study. But INFER-ACTION is not perfect. For example, the replacement of "rome" with "Rome" in the example above would be inferred as a substitution of only one letter, *i.e.*, *S*(18, 1, "R").

Recovering the Input Stream from Transcription Sequences

Although in the new *T*-sequence paradigm, method-specific keystrokes like BACKSPACE are no longer separately distinguished, the input stream (*IS*) can still be recovered if it is known that users behaved as they did in the old *IS* paradigm, *i.e.*, if users entered text sequentially at the end of the *IS* and BACKSPACE was their only form of error correction. Specifically, when there is only one character changed at the end of the *IS* with each action, we can simply recover the action by examining the difference between adjacent transcriptions. Note that substitution actions cannot directly be performed in the *IS* paradigm, only insertions and deletions at the end of the currently entered text. The entire *IS* can be rebuilt, as in the following example:

Transcription (T_i)		Action (A_i)	
T_0:	<null>	A_0:	<begin>
T_1:	`t`	A_1:	*I*(0, "t")
T_2:	`th`	A_2:	*I*(1, "h")
T_3:	`thw`	A_3:	*I*(2, "w")
T_4:	`th`	A_4:	*D*(2, 1)
T_5:	`the`	A_5:	*I*(2, "e")
IS:	**`thw<e`**		

Now that we have presented *T*-sequences, we show how they can be used to formulate the Incorrect and Fixed (*IF*) class and its separation into two subclasses, *IFc* and *IFe*.

EXTENSIONS TO THE INCORRECT-AND-FIXED CLASS

In text entry transcription studies, a common error correction behavior is to extensively use BACKSPACE, which often leads to deleting already-correct characters [20]. Consider this input stream:

```
quack<<<<uick
```

Above, "uack" is erased by four BACKSPACEs, and "uick" is added. However, the "u" was correct despite being backspaced. Similarly, the "ck" were correct despite being backspaced. Should they be counted as errors? To address this question, Soukoreff & MacKenzie [20] created new *IFc* and *IFe* subclasses of their Incorrect and Fixed (*IF*) class. *IFc* and *IFe* respectively stood for "incorrect-and-fixed correct characters" and "incorrect-and-fixed errors." This separation of *IF* into these two new subclasses enabled more accurate error rate calculations.

However, because *IFc* and *IFe* were based on the *IS* paradigm, prior work [20] assumed BACKSPACE was the only way to correct errors. In the more flexible *T*-sequence paradigm, a new algorithm is needed to separate *IF* into *IFc* and *IFe*. Therefore, we use a modified version of the Needleman-Wunsch algorithm [17]. As before, we assume no method-specific knowledge of the actions performing the correction of text.

Modified Needleman-Wunsch Algorithm for *IFc* and *IFe*

The Needleman-Wunsch algorithm [17] is a dynamic programming algorithm used to align biological sequences. The algorithm is more flexible than the algorithm used in the calculation of the minimum string distance (MSD) [21], *i.e.*, the Levenstheim string alignment algorithm [9].

In the Needleman-Wunsch algorithm, there are three types of character comparisons: *match, mismatch,* and *gap*, each of which are assigned scores during the alignment. *Match* means two characters are the same; *mismatch* means they are different; *gap* means one letter in one string lines up with a gap in the other string. For example, consider the alignment of "dynamic" and "plastic":

```
dyna-mic
-plastic
```

In this example, a '–' means a gap, of which there are two. There are also three matches and three mismatches.

We modified the original Needleman-Wunsch algorithm to not favor aligning string beginnings and endings by penalizing start and end gaps that occur after matches have been made while there are still characters left to match. This modification was necessary because in text entry transcription tasks, users try to align with the presented string (*P*). For example, if *P* = "true treasure" and *T* = "treasure", the unmodified Needleman-Wunsch algorithm would produce this:

```
true treasure
tr-e----asure
```

However, our modified algorithm produces this result:

```
true treasure
-----treasure
```

We also added a gap penalty to the algorithm, which assigns different penalties to the opening or extending of a gap. The purpose of this penalty is to promote the formation of connected gaps in the alignment, *i.e.*, favoring long contiguous gaps over disjointed short gaps. Consider aligning "Massachusetts" and "Massetts". If the penalty is the same for opening a new gap as it is for extending an existing one, then the result will be:

```
Massachusetts
Ma-s----setts
```

But if the penalty for opening a new gap is higher than for extending an existing one, the result will be:

```
Massachusetts
Mas-----setts
```

In our version of the Needleman-Wunsch algorithm, we set the score for *match* as +3, *mismatch* as -2, *opening* a new gap as -2, and *extending* an existing gap as -1.

Detecting *IFc* and *IFe* Characters

Our solution to finding correct (*IFc*) and erroneous (*IFe*) characters within the Incorrect and Fixed (*IF*) class [20] uses our modified Needleman-Wunsch algorithm [17]. Specifically, given a *T*-sequence, for each action A_i that is *delete* or *substitute*, we perform an alignment between T_{i-1} and *P*, the presented string; we then find the corresponding part Δ in *P* that aligns with the deleted characters in T_{i-1}. In the deleted substring, the characters that *match* Δ are in *IFc*, and other characters are in *IFe*. An example illustrates:

```
P:      All roads lead to Rome
T_{i-1}: All toads
T_i:    All                       A_i: D(4, 5)
```

The optimal alignment of P and $T_{i\text{-}1}$ is:

```
All roads lead to Rome
All toads-------------
```

Now, the part of P that aligns with the deleted characters "toads" is Δ = "roads". The "r" and "t" do not match, and are classified as errors in *IFe*. The "oads" suffix matches and is therefore classified as having correct characters in *IFc*.

NEW METRICS BASED ON TRANSCRIPTION SEQUENCES

To recap: thus far, we have shown how *T*-sequences can enable the calculation of error rates from the input stream (*IS*) paradigm of Soukoreff & MacKenzie [21,22]. These error rates are defined in Eqs. (3)-(5), above. We have also shown how to separate the Incorrect and Fixed (*IF*) class into two parts, one for characters that were erased and correct (*IFc*), and one for characters that were erased and erroneous (*IFe*) [20]. It should be clear by now that the *T*-sequence paradigm can produce the traditional error rate metrics from the *IS* paradigm. It should also be clear that the *T*-sequence paradigm can handle text input behaviors that occur outside the *IS* paradigm, *e.g.*, insertions, deletions, and substitutions *within* the transcribed text, the simultaneous entry or removal of multiple characters, and more.

Going further, the *T*-sequence paradigm can produce more than just the traditional error rates; it also gives rise to new metrics not formerly obtainable from the *IS* paradigm.

The new metrics that follow pertain to text entry transcription tasks with a presented string *P* and transcribed string *T*, just like for the traditional error rate metrics. Note that $|T|$ indicates the length of the final transcribed string. We begin with some basic count metrics, upon which we build.

Basic Count Metrics

Total Changed Characters (TCC) refers to the number of characters that change during the text entry process, including all characters that are inserted or deleted.

$$TCC = |T| + 2 \times IF \quad (6)$$

IF is added twice because any deleted characters were first inserted, constituting two changes per *IF* character.

Action Count (AC) refers to the number of actions taken during the text entry process. As described above, these actions are inferred from a *T*-sequence using INFER-ACTION. We also define counts for specific actions: the *Insertion Action Count (IAC)*, *Deletion Action Count (DAC)*, and *Substitution Action Count (SAC)*.

More specifically, ***Correction Action Count (CAC)*** gives the number of corrective actions, which are delete and substitute actions (*i.e.*, $CAC = DAC + SAC$). Similarly, ***Entry Action Count (EAC)*** gives the number of insert and substitute actions (*i.e.*, $EAC = IAC + SAC$), as both make new entries.

New Metrics

Characters per Action (CPA) shows the average number of characters changed per action:

$$CPA = TCC \,/\, AC \quad (7)$$

Characters per Correction (***CPC***) and ***Characters per Entry (CPE)*** convey how many characters are changed, on average, per correction or entry, respectively:

$$CPC = IF \,/\, CAC \quad (8)$$

$$CPE = (|T| + IF) \,/\, EAC \quad (9)$$

Action Efficiency (AE) conveys the number of characters one action can change in a given time period, *e.g.*, per second. It is therefore the "text-changing speed" of actions.

$$AE = TCC \,/\, \text{Total time} \quad (10)$$

More specifically, ***Correction Efficiency (CE)*** and ***Entry Efficiency (EE)*** indicate the "text-correcting speed" of actions and the "text-entering speed" of actions, respectively. *Correction time* refers to the total time of delete and substitute actions, and *Entry time* refers to the total time of insert and substitute actions.

$$CE = IF \,/\, \text{Correction time} \quad (11)$$

$$EE = (|T| + IF) \,/\, \text{Entry time} \quad (12)$$

Using the above metrics, we can categorize text entry methods into four types based on the effort to enter and correct text: (*i*) Easy entry, easy correction; (*ii*) Easy entry, hard correction; (*iii*) Hard entry, easy correction; and (*iv*) Hard entry, hard correction. Entry and correction difficulty is indicated by how much text can be added or deleted in one action, and by how fast it is to add or delete text. Therefore, the above *CPE* and *EE* metrics together are entry difficulty metrics; the *CPC* and *CE* metrics together are correction difficulty metrics. Following those, *CPA* and *AE* are overall difficulty metrics.

THE NEW EVALUATION TESTBED *TEXTTEST++*

The original *TextTest* tool [27] has been used to conduct numerous text entry studies (*e.g.*, [7,18,26]) and produce measures based on the input stream (*IS*) paradigm of Soukoreff & MacKenzie [21,22]. Inspired by *TextTest*, a Windows-based desktop application, we implemented *TextTest++*, a web-based testbed capable of running, logging, and analyzing text entry studies (Figure 3). By being web-based, *TextTest++* is platform-independent, capable of being utilized on any device that offers a web browser. *TextTest++* computes traditional metrics, including words per minute (*WPM*) and the error rates in Eqs. (3)-(5). It contains the algorithms described in this work, and produces all of the new metrics in Eqs. (6)-(12).

Figure 3 shows the main user interface for *TextTest++*. The program is written in JavaScript and logs each test in JSON format, which contains all of the *T*-sequences and inferred actions. A CSV file is also generated containing all traditional and new metrics described in this paper.

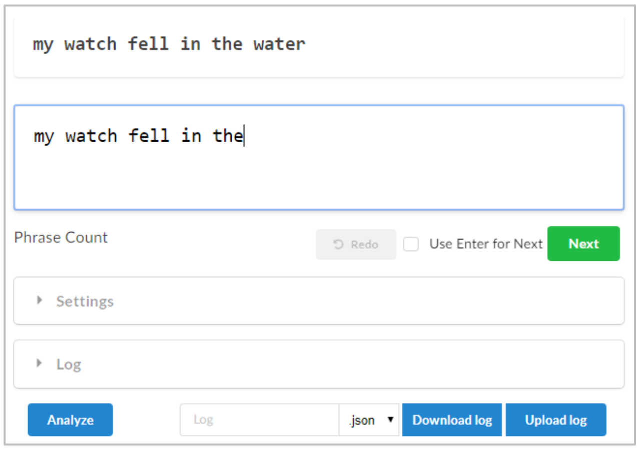

Figure 3. *TextTest++* is a new web-based text entry evaluation testbed that produces the traditional metrics from the *IS* paradigm and the new metrics from the *T*-sequence paradigm.

EXERCISING OUR ALGORITHMS, METRICS AND TOOL

We conducted an experiment to evaluate our algorithms, metrics, and the *TextTest++* testbed. To put these through their paces, we first tested three keyboard-based text entry methods: a laptop keyboard, an on-screen accessibility keyboard, and a smartphone keyboard. Our questions were:

1. Will the *T*-sequence paradigm yield the same results as the *IS* paradigm for transcriptions where the latter's experimental constraints happen to be upheld?
2. How well does the *T*-sequence paradigm handle modern text entry behaviors that might arise? How often do such behaviors arise?
3. What insights can our new metrics from the *T*-sequence paradigm provide?

Participants

We recruited 18 participants (15 female, 3 male) from our local university to partake in our study. Recruiting was done via flyers, emails, word-of-mouth, snowball sampling, and convenience sampling. Participants' ages ranged from 21 – 27. All participants were right-handed. They all indicated many years of experience typing on laptop keyboards and texting on smartphones. Participants were compensated $10 USD for 30 minutes.

Apparatus

We compared three keyboards: a laptop keyboard ("Laptop"), an on-screen desktop accessibility keyboard ("On-Screen"), and a smartphone touch keyboard ("Phone").

The Laptop keyboard was a Microsoft Surface Pro 4 typecover[3] measuring 11.60" × 8.54" × 0.20". Typing on such a keyboard is usually done serially, making it suitable for analysis with the *IS* paradigm—provided no arrow keys, no mouse, no copy/paste, no undo, and no word predictions are used.[4] Unlike in prior studies based on the *IS* paradigm, participants were free to employ such features.

The On-Screen desktop accessibility keyboard was operated using a Microsoft Wireless Mobile Mouse 4000 and the built-in Windows 10 On-Screen Keyboard. The keyboard measured 6.00" × 1.75". We chose this keyboard because by entering all text with the mouse, participants might use the mouse for other text entry behaviors, *e.g.*, to reposition the text cursor, or to drag-select over multiple characters. The keyboard did not have word prediction enabled.

The Phone touch keyboard ran on a Google Pixel smartphone measuring 2.74" × 5.66" × 0.33". The keyboard was *SwiftKey*,[5] a third-party smartphone keyboard equipped with opt-out auto-correction, word completion, a word prediction list, and, of course, the ability to reposition the text cursor with a tap or drag on the screen. The size of the SwiftKey keyboard itself was 2.74" × 2.60".

Our new *TextTest++* tool was used throughout the experiment (Figure 3). The textbox in *TextTest++* did not offer spell-check underlining. Timing for each phrase was from the first entered character to the last [12].

Procedure

For each participant, we randomly selected 30 phrases from a published text entry phrase set [14]. The ordering of phrases was randomized for each participant and method. Participants used each method in a fully counterbalanced order. At the start of using each text entry method, participants were given five warm-up phrases not included in the 30, which were the same for all participants and methods. Before the test phrases began, participants were told to proceed "as naturally as you can, while at the same time, keeping up your speed and accuracy." They were allowed to use the mouse in the Laptop condition, but not allowed to use the physical keyboard in the On-Screen keyboard condition. In the Phone condition, participants were told to use two-thumb typing. Participants were given 5 minutes of rest between each text entry method.

Design & Analysis

This experiment was a single-factor within-subjects design with three levels of text entry method: Laptop, On-Screen, and Phone. These levels were fully counterbalanced with 3! = 6 orders spread over 18 participants. Therefore, in all, we collected 18 participants × 3 methods × 30 phrases = 1620 text entry phrases.

Unlike most text entry experiments in which comparing the text entry methods is of primary interest, in this study, the methods were simply vehicles by which we could put our algorithms, metrics, and testbed through their paces.

[3] https://bit.ly/2LQCC7Q

[4] That's a lot of restrictions! Alleviating all of them (and more) is the precise benefit achieved by moving from the input stream model to the transcription sequence model.

[5] https://swiftkey.com/en/keyboard/android/

Therefore, we favor descriptive statistics over inferential statistics. (Indeed, we *expect* statistically significant differences among our text entry methods, but focusing on those differences would distract from our purposes here.)

RESULTS

In this section, we report the results of our study, validating our work's ability to produce traditional speed and error rate metrics, while going further to produce our new metrics. Please refer to Table 1 for the numeric results.

Words per Minute

We calculate words per minute (*WPM*) by subtracting the first character timestamp from the last character timestamp, being careful to define the length of the final transcribed string as one less than its character count (*i.e.*, $|T| - 1$) [12].

Uncorrected, Corrected and Total Error Rates

All uncorrected error rates (*UER*) were below 2%, indicating high accuracy of the final transcribed strings. Corrected error rates (*CER*) were clearly highest for the Phone, which was the most error-prone *during* entry.

Comparison to the Input Stream Paradigm

To compare our *T*-sequence paradigm to the *IS* paradigm, we extracted 50 random trials from the Laptop condition that happened to exhibit the strictly sequential editing process required by the *IS* paradigm. To find these compliant trials, we looked through *T*-sequences to find where only the last character in each step of the sequence was modified. This requirement was met by 98% of Laptop trials, 94% of On-Screen keyboard trials, but only 2% of Phone trials.

By translating *TextTest*++'s JSON files into the XML log file format required by the original *TextTest* program [27], we could use the log file analysis feature in the latter to produce the *C*, *INF*, and *IF* character classes. The counts produced by *TextTest* were identical to those produced by *TextTest*++, showing that *TextTest*++ and the *T*-sequence paradigm can subsume the *IS* paradigm correctly.

Separating *IF* into *IFc* and *IFe*

Recall that we also want to separate Incorrect and Fixed (*IF*) characters, *i.e.*, all erased characters, into *IFc* and *IFe*—those that were initially correct and initially erroneous, respectively [20]. Interestingly, for the Laptop and Phone methods, *IFc* > *IFe*, indicating that more initially correct characters were erased than erroneous ones. Such a result is consistent with observations of "pathologic error correction" [20], and also follows the use of the auto-correction feature on the Phone.

VALIDATING THE CORRECTNESS OF *INFER-ACTION*

Recall that we logged text cursor position changes to obtain ground truth actions for each text entry method. We can compare these ground truth actions to those inferred from our INFER-ACTION algorithm to see how well our algorithm performed (*i.e.*, when comparing T_{i-1} to T_i). Over our study's entire 1620 phrases, we found *no* differences between the results of INFER-ACTION and the ground truth information gleaned from text cursor position changes.

Furthermore, to ensure that INFER-ACTION correctly handles a variety of text entry methods, we conducted a follow-on study of three *more* text entry methods, none of which were keyboard-based: *Dasher* [25], *gesture typing* [8,30], and *T9* [11]. Dasher is a pointing-based continuous-motion zooming interface. Gesture-typing enables stroke-gestures with a finger or stylus atop a virtual keyboard, with gestures corresponding to entire words based on their shapes and the letter arrangements beneath them. T9 is a predictive text method for 12-key numeric keypads, where sequences of key-presses are progressively disambiguated to form the most likely words.

For Dasher, we used *Dasher* 5.0 with cursor speed set at 3.2. For gesture typing, we used *SwiftKey* in swiping mode. For T9, we used *Smart Keyboard Pro*. Dasher ran on the Microsoft Surface Pro laptop; the other two methods ran on the Google Pixel smartphone. Under the same configuration as the main experiment, 6 participants (2 female, 4 male, ages 23 – 26) each transcribed text for 10 minutes with each method. The condition order was fully counterbalanced. Before the formal study, each participant learned about each method and took ~20 minutes total to practice. Participants were paid $15 USD.

We collected 58 phrases with Dasher, 202 phrases with gesture typing, and 111 phrases with T9. Among the total 371 phrases, results generated from INFER-ACTION and results using ground truth cursor movement were exactly the same. This result indicates that INFER-ACTION works well across quite different text entry methods, including those that enter entire words at once.

DISCUSSION

Our experiment produced several findings. As we suspected, the different text entry methods resulted in different user behaviors. For example, with the Laptop keyboard, BACKSPACE was almost exclusively used for error correction, but with the On-Screen keyboard, mouse-based error correction with cursor repositioning was often used, a behavior formerly prohibited by the input stream (*IS*) paradigm. Even further, one participant (P6) used copy/paste in two phrases in the On-Screen keyboard condition. When P6 typed the phrase "the dreamers of dreams," she drag-selected the first "dream", copied it, and then pasted it at end of the phrase, finally typing an "s". This behavior, too, was prohibited in the *IS* paradigm, but now can be supported.

Examining *Characters per Action (CPA)* and *Action Efficiency (AE)* together creates a picture of input difficulty. The Laptop and On-Screen keyboards had *CPA*s of 1.000 and 1.002, respectively, but the Laptop's *AE* is much higher than that of the mouse-driven On-Screen keyboard: 5.894 *vs.* 1.586 actions per second. This difference explains the faster entry speed of the Laptop keyboard. The Phone had the largest *CPA* and *AE*, indicating that it is even easier to act upon text than with the Laptop keyboard. This ease arises, for example, when one types only the initial characters of a word, and then word completion finishes the rest.

Input Method	Words per Minute (WPM)	Uncorrected Error Rate (UER)	Corrected Error Rate (CER)	Total Error Rate (TER)	Incorrect and Fixed – Correct (IFc)	Incorrect and Fixed – Errors (IFe)
Laptop	65.35 ± 28.19	0.003 ± 0.003	0.046 ± 0.031	0.049 ± 0.305	1.17 ± 0.83	0.39 ± 0.41
On-Screen	17.85 ± 2.22	0.004 ± 0.003	0.033 ± 0.015	0.037 ± 0.016	0.33 ± 0.19	0.73 ± 0.36
Phone	33.71 ± 7.92	0.013 ± 0.008	0.388 ± 0.059	0.401 ± 0.056	16.38 ± 3.25	3.34 ± 2.78

Input Method	Characters per Action (CPA)	Characters per Correction (CPC)	Characters per Entry (CPE)	Action Efficiency (AE)	Correction Efficiency (CE)	Entry Efficiency (EE)
Laptop	1.00 ± 0.00	1.003 ± 0.00	1.00 ± 0.00	5.89 ± 2.47	2.98 ± 0.91	6.11 ± 2.60
On-Screen	1.00 ± 0.00	1.00 ± 0.00	1.00 ± 0.00	1.59 ± 0.20	0.90 ± 0.14	1.63 ± 0.21
Phone	1.89 ± 0.07	2.96 ± 0.44	1.70 ± 0.14	6.55 ± 2.08	6.53 ± 4.19	7.22 ± 1.80

Table 1. Means and standard deviations for speed, error rates, *IFc* and *IFe* counts, and our new metrics arising in the *T*-sequence paradigm. *CPA*, *CPC*, and *CPE* are counts. *AE*, *CE*, and *EE* are counts per second. See Eqs. (7)-(12).

We now turn to entry and correction difficulty. *Characters per Entry (CPE)* and *Characters per Correction (CPC)* are nearly ~1.00 for the Laptop and On-Screen keyboards. But interestingly, *Entry Efficiency (EE)* is about twice as high as *Correction Efficiency (CE)* for both keyboards, indicating that deleting a character takes about twice as long as entering it in the first place.

Of note is that participants deleted more correct characters than erroneous ones with the Laptop keyboard (*IFc* > *IFe*). With auto-correct active on the Phone keyboard, it is no surprise that this occurred, but that it occurred on the Laptop keyboard supports informal observations of "pathologic error correction" from prior work [20]. The *EE* of the Laptop keyboard was quite high (6.111), so perhaps participants did not mind if they had to re-type correct characters after backspacing through them to reach an incorrect one. The situation was different with the On-Screen keyboard, whose *EE* was only 1.629, and whose *IFc* < *IFe* (0.33 *vs.* 0.73).

Limitations

Indeed, the *T*-sequence model has limitations: (1) Because *T*-sequences focus on providing general metrics across different text entry methods, gaining insights about *how* a text input method does its work is not possible with the model, just as it was not possible with the *IS* model. Evaluators wishing to examine particular features of a text entry method must still build a custom evaluation tool. (2) Some text entry methods such as T9 produce temporary characters during input, with those temporary characters often appearing in-place. If such characters are actually *committed* into the textbox, then additional characters will be counted in *IF*. This problem, however, does not arise if these temporary characters are not actually committed until they are resolved; for example, temporary characters might appear in a separate list from which the user makes a selection. (3) The metrics associated with the *T*-sequence model are still character-level metrics, even though the model itself and its associated actions accommodate word-level behaviors such as those used in gesture typing. For word- or phrase-level input methods such as voice typing or gesture typing, INFER-ACTION still produces correct character-level metrics, as was validated in our second study, but the relevance of these metrics to word-level methods might be less. (4) In our experiment, INFER-ACTION worked flawlessly, but the text entry behaviors of participants in real life might differ from participants' behaviors in a lab-based transcription study. And we know INFER-ACTION is not perfect. (Recall the "rome" and "Rome" example, above).

CONCLUSION

Although the input stream (*IS*) paradigm of text entry evaluation has been highly successful, modern text entry methods require more flexible evaluation paradigms. To achieve this, we presented a method-independent paradigm based on transcription sequences, or *T*-sequences, which contain transcription snapshots after every text-changing action occurs. We built an abstraction of actions and showed how to infer these actions from transcription string pairs. We also showed how the traditional character classes used in error rate measurement [20,22] can be calculated in the *T*-sequence paradigm. Furthermore, we presented new metrics arising from this paradigm. Our study demonstrated that *T*-sequences can supersede the *IS* paradigm and offer new insights not formerly possible, while greatly lessening the constraints on the evaluation and supporting many modern text entry behaviors. We offer our new method-independent web-based tool, *TextTest++*, to researchers and practitioners in the hope that text entry evaluations will be made easier, more flexible, more realistic, and more informative.

ACKNOWLEDGEMENTS

This work was supported in part by the University of Washington Information School. Any opinions, findings, conclusions or recommendations expressed in our work are those of the authors and do not necessarily reflect those of any supporter.

REFERENCES

[1] Ahmed Sabbir Arif and Ali Mazalek. 2016. WebTEM: A web application to record text entry metrics. *Proceedings of ISS 2016*. New York: ACM Press, 415–420. DOI: 10.1145/2992154.2996791

[2] Ahmed Sabbir Arif and Wolfgang Stuerzlinger. 2010. Predicting the cost of error correction in character-based text entry technologies. *Proceedings of CHI 2010*. New York: ACM Press, 5–14. DOI: 10.1145/1753326.1753329

[3] Abigail Evans and Jacob O. Wobbrock. 2012. Taming wild behavior: The Input Observer for obtaining text entry and mouse pointing measures from everyday computer use. *Proceedings of CHI 2012*. New York: ACM Press, 1947–1956. DOI: 10.1145/2207676.2208338

[4] Leah Findlater, Jacob O. Wobbrock and Daniel Wigdor. 2011. Typing on flat glass: Examining ten-finger expert typing patterns on touch surfaces. *Proceedings of CHI 2011*. New York: ACM Press, 2453–2462. DOI: 10.1145/1978942.1979301

[5] Jun Gong and Peter Tarasewich. 2006. A new error metric for text entry method evaluation. *Proceedings of CHI 2006*. New York: ACM Press, 471–474. DOI: 10.1145/1124772.1124843

[6] Dietrich Klakow and Jochen Peters. 2002. Testing the correlation of word error rate and perplexity. *Speech Communication 38* (1–2), 19–28. DOI: 10.1016/S0167-6393(01)00041-3

[7] Thomas Költringer, Poika Isokoski and Thomas Grechenig. 2007. TwoStick: Writing with a game controller. *Proceedings of Graphics Interface 2007.* Toronto: Canadian Information Processing Society, 103–110. DOI: 10.1145/1268517.1268536

[8] Per-Ola Kristensson and Shumin Zhai. 2004. SHARK^2: A large vocabulary shorthand writing system for pen-based computers. *Proceedings UIST 2004*. New York: ACM Press, 43–52. DOI: 10.1145/1029632.1029640

[9] Vladimir I. Levenshtein. 1966. Binary codes capable of correcting deletions, insertions, and reversals. *Soviet Physics Doklady 10* (8), 707–710.

[10] James R. Lewis. 1999. Input rates and user preference for three small-screen input methods: Standard keyboard, predictive keyboard, and handwriting. *Proceedings of the Human Factors and Ergonomics Society 43rd Annual Meeting*. Santa Monica, CA: Human Factors and Ergonomics Society, 425–428. DOI: 10.1177/154193129904300507

[11] Nuance, LLC. 2019. T9 – The global standard for mobile text input. Retrieved July 12, 2019 from https://bit.ly/32kl4GN

[12] I. Scott MacKenzie. 2015. A note on calculating text entry speed. Retrieved July 12, 2019 from https://www.yorku.ca/mack/RN-TextEntrySpeed.html

[13] I. Scott MacKenzie and R. William Soukoreff. 2002. A character-level error analysis technique for evaluating text entry methods. *Proceedings of NordiCHI 2002*. New York: ACM Press, 243–246. DOI: 10.1145/572020.572056

[14] I. Scott MacKenzie and R. William Soukoreff. 2003. Phrase sets for evaluating text entry techniques. *Extended Abstracts of CHI 2003.* New York: ACM Press, 754–755. DOI: 10.1145/765891.765971

[15] I. Scott MacKenzie and Shawn X. Zhang. 1999. The design and evaluation of a high-performance soft keyboard. *Proceedings of CHI 1999*. New York: ACM Press, 25–31. DOI: 10.1145/302979.302983

[16] Edgar Matias, I. Scott MacKenzie, and William Buxton. 1996. One-handed touch typing on a QWERTY keyboard. *Human-Computer Interaction 11* (1), 1–27. DOI: 10.1207/s15327051hci1101_1

[17] Saul B. Needleman and Christian D. Wunsch. 1970. A general method applicable to the search for similarities in the amino acid sequence of two proteins. *Journal of Molecular Biology 48* (3), 443–453. DOI: 10.1016/0022-2836(70)90057-4

[18] Tao Ni, Doug Bowman, and Chris North. 2011. AirStroke: Bringing unistroke text entry to freehand gesture interfaces. *Proceedings of CHI 2011*. New York: ACM Press, 2473–2476. DOI: 10.1145/1978942.1979303

[19] Sherry Ruan, Jacob O. Wobbrock, Kenny Liou, Andrew Ng, and James A. Landay. 2017. Comparing speech and keyboard text entry for short messages in two languages on touchscreen phones. *Proceedings of IMWUT 1* (4). New York: ACM Press. Article No. 159. DOI: 10.1145/3161187

[20] R. William Soukoreff and I. Scott MacKenzie. 2004. Recent developments in text-entry error rate measurement. *Extended Abstracts of CHI 2004*. New York: ACM Press, 1425–1428. DOI: 10.1145/985921.986081

[21] R. William Soukoreff and I. Scott MacKenzie. 2001. Measuring errors in text entry tasks: An application of the Levenshtein string distance statistic. *Extended Abstracts of CHI 2001*. New York: ACM Press, 319–320. DOI: 10.1145/634067.634256

[22] R. William Soukoreff and I. Scott MacKenzie. 2003. Metrics for text entry research: An evaluation of MSD and KSPC, and a new unified error metric. *Proceedings of CHI 2003*. New York: ACM Press, 113–120. DOI: 10.1145/642611.642632

[23] Dan Venolia and Forrest Neiberg. 1994. T-Cube: A fast, self-disclosing pen-based alphabet. *Conference Companion of CHI 1994*. New York: ACM Press, 265–270. DOI: 10.1145/191666.191761

[24] Keith Vertanen, Haythem Memmi, Justin Emge, Shyam Reyal, and Per-Ola Kristensson. 2015. VelociTap: Investigating fast mobile text entry using sentence-based decoding of touchscreen keyboard

input. *Proceedings of CHI 2015*. New York: ACM Press, 659–668. DOI: 10.1145/2702123.2702135

[25] David J. Ward, Alan F. Blackwell and David J. C. MacKay. 2000. Dasher—a data entry interface using continuous gestures and language models. *Proceedings of UIST 2000*. New York: ACM Press, 129–137. DOI: 10.1145/354401.354427

[26] Andrew D. Wilson and Maneesh Agrawala. 2006. Text entry using a dual joystick game controller. *Proceedings of CHI 2006*. New York: ACM Press, 475–478. DOI: 10.1145/1124772.1124844

[27] Jacob O. Wobbrock and Brad A. Myers. 2006. Analyzing the input stream for character-level errors in unconstrained text entry evaluations. *ACM Transactions on Computer-Human Interaction 13* (4), 458–489. DOI: 10.1145/1188816.1188819

[28] Xin Yi, Chun Yu, Weijie Xu, Xiaojun Bi and Yuanchun Shi. 2017. COMPASS: Rotational keyboard on non-touch smartwatches. *Proceedings of CHI 2017*. New York: ACM Press, 705–715. DOI: 10.1145/3025453.3025454

[29] Xin Yi, Chun Yu, Mingrui Zhang, Sida Gao, Ke Sun and Yuanchun Shi. 2015. ATK: Enabling ten-finger freehand typing in air based on 3D hand tracking data. *Proceedings of UIST 2015*. New York: ACM Press, 539–548. DOI: 10.1145/2807442.2807504

[30] Shumin Zhai and Per-Ola Kristensson. 2012. The word-gesture keyboard: Reimagining keyboard interaction. *Communications of the ACM 55* (9), 91–101. DOI: 10.1145/2330667.2330689

[31] Suwen Zhu, Tianyao Luo, Xiaojun Bi and Shumin Zhai. 2018. Typing on an invisible keyboard. *Proceedings of CHI 2018*. New York: ACM Press. Paper No. 439. DOI: 10.1145/3173574.3174013